ASAP : a protocol for symbolic computation systems

Stéphane Dalmas, Marc Gaetano, Alain Sausse

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

ASAP
a protocol for symbolic computation systems

Stéphane Dalmas

Marc Gaëtano

Alain Sausse

N° 162

Mars 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

1994

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

ASAP
a protocol for symbolic computation systems

Stéphane Dalmas
Marc Gaëtano*
Alain Sausse**

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet SAFIR

Rapport technique n°162 — Mars 1994 — 31 pages

Abstract: This report describes the conception and the implementation of a simple protocol for exchanging mathematical data between processes.

Our purpose is to enable symbolic computation systems to communicate and cooperate together but also with user programs, graphical interfaces, curve and surface plotters.

In the first part, we study the general problem of the exchanging of mathematical data. The second part defines the protocol. Then, we present the documentation of a C library which implements ASAP.

Key-words: symbolic computation, mathematical data, scientific computing, protocol

(Résumé : tsvp)

Partially supported by ESPRIT/BRA project number 6846 : POlynomial System SOlving (POSSO)

*I3S, CNRS URA 1376,Université de Nice Sophia Antipolis
**Laboratoire de Mathématiques, URA 168, Université de Nice Sophia Antipolis

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

ASAP
un protocole pour des systèmes de calcul formel

Résumé : Ce rapport décrit la conception de l'implémentation d'un protocole simple permettant l'échange de données mathématiques entre deux processus.

Notre but est de faire communiquer des systèmes de calcul formel entre eux mais aussi avec des programmes utilisateurs, des interfaces graphiques, des traceurs de courbes et de surfaces, etc ...

Dans la première partie, on étudie le problème général de l'échange de données mathématiques. La deuxième partie définit le protocole. Enfin, on présente la documentation d'une bibliothèque de fonctions C qui implémente ASAP.

Mots-clé : Calcul formel, objets mathématiques, calcul scientifique, protocole

Introduction

There is a large family of systems for symbolic computation. Most of them have libraries of algorithms for the resolution of complex mathematical problems. From general purpose systems to specialized packages, the user can find the suitable tools to solve his problems. Nevertheless, in many cases the user wants to deal with more than one of these tools at once to get the best of them all. There is no simple way to make systems collaborate in this way except using files and ad-hoc tools to convert data from one system to another. There is also an increasing need for using symbolic computation as part of existing applications, in a non interactive way (for example in CAD or constraint-based systems) or to connect such a system to a graphical user interface or to sophisticated plotting programs.

So, we need tools that can make mathematical softwares exchange symbolic data more easily. Some attention has been paid to this problem in the last few years. Some commercial softwares provide their own solution for communication (like Mathematica with MathLink or the C-callable version of Maple). More recently, there has been a first attempt to define a "vendor independent" standard protocol for symbolic computation with the *OpenMath* proposal. In the future, our work on ASAP is intended to be merged with this standard.

The communication between mathematical softwares implies (as a first step) a common format to exchange mathematical data. There are various possible formats (encodings) for exchanging data : binary, ASCII, *etc*, depending on the trade-off between efficiency and machine/application independence. The protocol we propose, named ASAP, is based on the hypothesis that the overhead of the communication (encoding, emitting, decoding) is expected to be rather small compared to the time spent in the systems which communicate.

Mathematical objects are represented quite naturally by terms, which are commonly represented as trees. To provide more flexibility, we use attributed trees as the data structure of ASAP. An attributed tree is a tree such that each node carries one or more attributes. An attribute is a name possibly associated with another tree (its value). An attribute could be used to carry additional mathematical information (mathematical domains ...) or control information (description of subterms sharing, termination, interrupt handling...).

There is no predefined meanings given to operators in ASAP. Only a small set of "reserved" attributes have special meanings and are interpreted directly by the protocol. The two communicating processes must then agree on a common meaning for the operators. Such conventions can naturally lead to the definition of an appropriate "subprotocol".

Part 1

Designing a protocol for exchanging mathematical objects

1.1 Our goals

The general goal is to allow applications to exchange mathematical data with an emphasis on *symbolic* objects (*non-numeric* data).
Some very basic questions should be answered in this context :

- What exactly are these data ?
- How do applications use these data ?
- What does "exchanging" exactly means ?

We address the first two questions in the following sections. The last question could be answered now. The kind of exchange we have in mind is related to interprocess communication, where two running programs work together. We may express this using the "client-server model" : at some point a program (the client) needs to perform a computation and delegates it to a server. For example, a graphical user interface can be viewed as a client which does not perform any computation by itself. We are interested in the methods to achieve such functionalities. Another kind of exchange we can think of could be through a persistent medium (typically a file).

This is not the main issue here although part of what we are going to say and propose can be used to store mathematical objects on disks.

Some other questions could be asked in the more precise context of interprocess communication, that have a more technical flavor (and that can heavily depend on the target operating system):

- How is the communication established ? What form of addresses are we going to use ?
- How are exceptional conditions handled (for example, interruption requests or unexpected termination of a process) ?

We will try to answer some of the above questions in the following and provide some insights on the design of a protocol to exchange mathematical objects and the various choices that can be made.

We will work on the assumption that this protocol will be used for situations where transmission time is negligible compared with the time spent in computing by the two communicating processes. Hence, emphasis will be on simplicity and adaptability rather than efficiency.

1.1.1 Communicate, what for ?

There are two typical examples which show the need to communicate mathematical data:

- a symbolic computation system needs to communicate with its interface when they are working as two separate processes (like in *Maple* and *Mathematica*).
- a program needs to have some kind of symbolic computation done. The aim is to issue some "computation requests" within the program and get a result back.

The first example does not really match our claim of exchanging mathematical data. The involved objects are more "data structure" oriented, and in fact have no clear mathematical meaning. The interface is just a kind of "fancy" parser and a parse tree is the object that is exchanged. The system sends an "output form" to be displayed as a result. None are really mathematical objects: they are not the kind of objects manipulated by mathematicians or engineers, but nevertheless we must be able to deal with that too.

The second example is much more interesting from a mathematical point of view because the request is supposed to have a clear mathematical meaning. It shows that in practise we need more than just means to exchange mathematical data. We should be able to express computation requests and handle some error/failure indication to distinguish between normal and abnormal results.

Computer algebra systems do not usually make a distinction between a computation request (what will cause a computation and produce a result) and a mathematical data which should not be subject to any kind of "evaluation" (we don't consider basic simplifications as an evaluation in this case).

1.1.2 What kind of mathematical data ?

Certainly, we should cover all the range of what is manipulated by a computer algebra system. This includes integer, rational numbers, rational approximations of reals ("big floats"), algebraic numbers, p-adic numbers, transcendental numbers, polynomials, rational functions, functions as "expressions", functions defined piecewise, matrices with various kinds of entries, finitely presented groups... It is difficult to make such a list exhaustive.

What we need to define is an encoding for all these objects. Obviously we have two choices:

- a very semantical coding : the representation clearly defines the kind of mathematical object, independently of any context.

– an "ambiguous" coding : we have to rely on a kind of context to know the correspondence between the representation and the associated mathematical object. Such a context can be : I know that I communicate with a *Maple* server, so this term with an `array` operator is a matrix...

The extreme variety of types of mathematical objects (in fact an infinity, if we take parametrized constructs into account...) makes us naturally believe that an ambiguous coding is the correct solution.

Of course, the simplest solution is to encode the objects as terms (or trees in a more concrete setting). If we allow our terms to be attributed (each operator can carry a set of pairs made of a name and a value that can be another term) we can express additional information with no loss of simplicity for the representation.

Note that all general purpose systems (with the exception of *Axiom*) tend to use that sort of encoding (terms with no attributes), although they don't use "pure" terms. They also have constructors for lists and arrays. We will argue about the advantages of such a richer representation in a following section.

1.1.3 What kind of communication ?

From a very pragmatic point of view, the programmer should be able to send data to a particular server or client (in the sequel we will say "partner") that may be running on a remote machine and retrieve a result (or an answer). In practice, this will be done through a set of functions or procedures that will provide the programming interface to the protocol . Of course, we want the communication (as seen from the programmer) to be reliable (no alteration, no duplication...) and with no arbitrary limitation (like a maximum size for objects). The user certainly needs a way to find the required partner: each partner should thus have a particular address. The communication can be "connection oriented", meaning that prior to any exchange of data the programmer has to establish a connection with the needed partner (in the same way that before writing to a file he should obtain a "file handle" by opening the file). We can also think of a "connection-less" communication where there is no connection prior to communication (the address of the partner is included in each request). This kind of communication can be well adapted for RPC-like situations where there is no need to known which partner takes care of the remote computation.

The way the communication is established is irrelevant to the protocol used to encode data. This is only partially true because it can induce constraints on the protocol and we should be aware of that. For example, if the communication is connection-less we should not expect our partner to remember anything from one request to another. In this case it doesn't make much sense to ask for an initialization or a termination of the partner.

1.1.4 What we should provide as a "standard"

From what has been said before, a useful standard for exchanging mathematical objects should provide :

- a "high-level" description of what will be exchanged (a kind of abstract data type definition or abstract syntax of the transmitted objects).
- a "low-level" encoding used to translate the high-level objects to streams of bytes (which are the ultimate objects that are exchanged).
- the control part of the protocol to describe the overall organization of the communication (connection oriented or connection-less, handshaking, termination, interruption...)

Additional tools that could be provided and that are more operating system or programming language dependent :

- a naming convention for partners i.e. the kind of addresses that can be used to communicate
- an API (Application Programming Interface) to access the protocol for each relevant programming languages (at least C and also Common Lisp as they are the most used languages in our field...)
- a set of software tools to ease the development of servers or clients (partial automatic generation of the code for communication)

1.2 Encoding of attributed trees

As attributed tree is the abstraction we choose to build our protocol, let us compare some possible encodings. Two possible choices are a "lisp-like" representation based on s-expressions (that is the representation in open math proposal, see [Vorkoetter]) and a prefix form.

1.2.1 The ASAP choices

ASAP defines an encoding for attributed trees. ASAP means "A Simple ASCII Protocol" mainly because it uses simple lexical conventions to suppress the need of most tags. Here is, roughly, the encoding used by ASAP :

- a term whose operator is *op* with no attributes is represented as the characters of *op* followed by a white space, the number of subterms, a white space and the subterms themselves, separated by spaces.
- a term with attributes (pairs of an attribute name and the value of the attribute which is itself a term) is represented as a special tag (one byte), the name of the operator, a space, the number of attributes followed by a space and the pairs of the attribute names and their values separated by spaces followed by the subterms.
- leaves can be integers or binary data (a sequence of arbitrary bytes beginning with a special tag followed by the size, followed by a space and the data themselves ...)

All integers (leaves as well as numbers of attributes, subterms or size of binary data) are coded as the sequence of their figures (in base 16 or 10). This is quite efficient in the case of a small number of subterms or attributes and places no limitation on the size of binary data.

One advantage of ASAP is the fact that "binary data" can be sent very easily (no need to escape anything). This can be use to "encapsulate" more efficient protocols or encoding transparently inside ASAP.

1.2.2 Comparison with a "lisp-like" encoding

The encoding described in the "open math proposal" (that will be referred as "open math" in the sequel) can be qualified as lisp-like. The data manipulated by open math are basically the same as ASAP, namely attributed trees. The main difference is in the "concrete" syntax similar to Lisp which is more "human oriented".

Although there is no major differences between the ASAP and open math encoding, such a "human oriented" syntax has some drawbacks:

- we need "escape characters" to send strings or binary data (to prevent the interpretation of delimiters). We cannot blindly send a chunk of bytes, we need a conversion. It thus prevents the efficient encapsulation of a specialized encoding.
- the lexical conventions for operators are more restricted or need escape characters
- the number of subterms is not know until the end of the term is reached

1.2.3 Comparison with a richer encoding

We can think of using a richer encoding i.e. one using more types of nodes than ASAP, for example, nodes for sets or lists, or even polynomials... Such an encoding can be trivially emulated with terms (and attributes) at a cost which is obviously not prohibitive. We think that there is no real need for a richer representation for the following reasons:

- the existing systems mostly operate on terms
- other data structures (like lists or arrays) are mostly used to build bunches of terms. They can be encoded themselves by terms with no real loss of efficiency if they are not the "main" type of data we need to exchange
- having terms makes the protocol simpler and more homogeneous, and consequently encoding and decoding terms is simpler.
- semantically, a richer representation does not give much i.e. the expressivity is not really enhanced in this context where the exact meaning of a term is dependent on the context of the communication anyway.

Of all these reasons, the main is maybe to keep the protocol as simple as possible.

1.3 The control part of the protocol

Here, there are some choices that need to be made. Some basic questions must be answered :

- Do we need connection oriented or connection-less communication ?
- Should we have a kind of "handshaking" when initiating a connection ?
- Should we require an acknowledgment for each request ?

– Should we have a way to detect the termination of the partner in the case of a "connection-oriented" communication ? Should we have a timeout ?

For the kind of applications that we have in mind, a connection oriented model of communication is certainly the most adequate. But of course most of what we will say in the following can be applied to a connection-less style.

As we should keep the protocol simple we do not think we should require any sort of handshaking or acknowledgment.

For handshaking, the great variety of partners makes difficult to find something interesting enough to exchange before any useful communication can take place. For acknowledgment, we work on top of a reliable lower level protocol. Furthermore, it is clear that a request can have an arbitrary length and thus take an arbitrary time just to be parsed. These two observations limit the interest of systematic acknowledgment.

There is a real need to be able to interrupt because of the potential length of the computations and to send and receive some "out of band" data. ASAP uses two communication channels, one for normal communication and the other for "out of band data" like exceptional conditions (including requests to interrupt).

1.4 Expressing computation requests and the semantics of operators

The problem here is to be able to abstract from our most common examples (the ones of computer algebra systems) and find a set of minimal assumptions on the behaviour of a "computing partner".

If we need to have some computation done by an "external" partner there are two cases to be considered:

– we know exactly what partner will handle our request (for example *Maple*)
– we don't care as long as the work is done

In the first case, we are just a simple Maple user and the request can be expressed in an "abstract" form of the Maple syntax. In the second case, the situation is more complicated because the request should be as system independent as possible. In fact, it cannot be truly system independent (as some possible partners cannot handle the desired requests) but we should require as few knowledge as possible from the user.

Practically, this means that the following conditions should be met:

– we must be able to express exactly what is going to be computed. If the user wants to differentiate a given expression containing a functional expression like `factor(...)` a server should not try to evaluate (factorize) the argument of `factor` unless explicitly asked for.
– the interpretation of the "non computed" arguments to a computation request should not surprise the user. For example, if one wants to differentiate an expression containing something like `W(x)` and that the symbol `W` as no special meaning for the user, the server should not impose any meaning by itself.

The first condition can be easily met. Exchanging attributed terms makes it easy to express computation requests by using a particular attribute.

The second condition needs some kind of convention that assigns mathematical meanings to some operators. For example, `sin` can denote the mathematical sine function (although it is not so easy, think of the difference between the sine function in the real and complex domain...) Anyway we must also provide a way (through attributes) to say "this has no meaning for me, so it must have no meaning for you either" (if we don't want the user to browse through all the documentation to find which operators have meanings).

Part 2

The ASAP Protocol : a description

2.1 The model of communication

The basic intended model of communication is simple : a client program requests computations from a server program by sending it messages. The server computes an answer and send it back to the client. All the requests and answers consist of a sequence of bytes encoding a term (an attributed tree). It is not required that ASAP terms are received (or sent) as atomic units as the low-level i/o facilities probably uses fixed-size buffers.

ASAP is quite simple as a communication protocol because :

- it assumes that the communication is reliable. All the bytes sent to one process will arrive unduplicated, in-order, unaltered to the other.
- it does not enforce any kind of negotiation between the two processes like a particular sequencing of messages or acknowledgements.

ASAP does not require neither that a server has to wait for an answer of the client before sending another message nor that the server have to answer at all. It is the responsibility of the client and the server to agree on the exact details of their interoperation. It is expected that a server will have a published interface documenting what it does in response to which messages. This will define the semantics of the messages (the meaning of the terms).

The communication uses two independent full-duplex byte-stream connections (channels). The two communicating processes write and read bytes to and from these two channels. One channel is used for normal communications (the normal channel), the second one is used for exceptional communications (the urgent channel). These exceptional events can include interrupt requests (the client wants to interrupt a running client), status information (a running client periodically sends a message describing the state of the computation) *etc* ... It is expected that a process will be able to react within a short time delay to any message sent to the urgent channel. The process should be able to interrupt a running computation, read and interpret the data on the urgent channel, take whatever actions it implies and resume its computation if applicable. If resuming is desired the communication must only use the

urgent channel before the resuming (and not the normal channel). The urgent channel should not be used to signal "synchronous" errors (errors as responses).

In the sequel we will assume that the connection uses two full-duplex channels. The current implementation allows to replace a full-duplex channel with a pair of channels (one channel for reading and the other for writing) as the input-output facilities found in many languages do not allow full-duplex (read/write) channels.

From now we use bytes and characters as synonymous. ASAP uses the ASCII encoding of characters to map usual characters to 8 bits bytes.

A client can have any number of servers. Each communication requires two channels (there is no "sharing" of channels). From the point of view of the protocol, there is no difference between a client and a server process.

A client can differ from a server by the way it handles certain requests. For example, a server will certainly react to a termination or interrupt request sent by a client. The converse is likely not to be true.

2.2 Establishing a connection

ASAP does not specify the way a connection is established between a client and a server. It assumes that two channels are available. This section contains some hints on how such a connection can take place.

The "target machines" of ASAP run operating systems that supports the "socket" interprocess communication mechanism. ASAP should use sockets of type SOCK_STREAM providing sequenced, reliable 2-way-connection based full-duplex byte streams.

The urgent channel could be implemented by enabling the `SIGIO` signal on the file descriptor corresponding to the urgent channel (this particular signal will be sent everytime something arrive on the urgent channel, so that the current computation will be interrupted and control passed to an appropriate signal-handler). The out-of-band data mechanism presents on SOCK_STREAM socket is not appropriate for this purpose (there is a very small and implementation dependent limit on the number of bytes that can be transmitted).

For two processes to communicate each one must have a way to "find" the other. This can be done if the two processes are specially related (like for the pipe facility between a parent and a child in UNIX) or more generally by means of particular addresses uniquely identifying each process. An interesting problem is then how to obtain such addresses... There is a library of C functions providing a convention for that matter and easy ways to launch clients. Services are identified with symbolic names and the correspondence between a name and a program is done through a configuration file (in fact, a set of such files is searched, each user can have its own file and also access other people files). The services can be specified as remote (on a particular machine) or local, using a "dynamic" negotiated address or a static one.

2.3 The terms

The main purpose of ASAP is to encode a mathematical term as a sequence of bytes to be sent and received on the two channels. What we call terms or attributed trees in this document are defined (recursively) as follows : A term is either

- a leaf consisting of a basic object
- an operator (a string) and a set of pairs consisting of a string (an *attribute*) and a term (the attribute value), and a list of terms (the subterms or sons of the operator).

Terms can be represented as directed, connected acyclic graphs whose nodes are labeled by strings called *operators* and sets of attributes. The leaves of our terms can be operators with no sons, integers and arrays of bytes.

It should be noted that contrary to some common conventions an arity is not attached to an operator. `A(A(A(), 1))` where `A` is used with arities 0, 1 and 2 is a perfectly valid term to be transmitted (and received) by ASAP.

Here are some trivial examples of terms written in a common concrete syntax :

```
A(B(1, C(123, D())))    +(X(), Y(), 3)
```

For attributed terms, we can use a list after the name of the operator (it is just a convenient notation to avoid drawing trees...) :

```
A(B[(a, 2), (b, E(B[(a, A())]()))](1, C(123, D())))
```

2.4 Encoding terms

Here is a pseudo-code (whose meaning should be straightforward...) describing the encoding of a term :

```
procedure output_term (t) is
if integer? (t) then
  output_integer (t)
  output_white()
else if binary? (t) then
  output_binary (t)
  output_white()
else
    if attributed? (t) then
        output_attribute_tag()
        output_operator (operator (t))
        output_white()
        output_integer (number_of_attributes(t))
        output_white()
        output_integer (arity t)
        for i in [1..number_of_attributes (t)] do
            output_attribute (attribute_name (t, i))
```

```
            output_white()
            output_term (attribute_value (t, i))
            output_white()
        end for
    else
        output_operator (operator t)
        output_white()
        output_integer (arity t)
        output_white()
    end if
    for i in [1..number_of_sons (t)] do
        output_term(son(t, i))
    end for
end if
end output_term
```

The function output_white() sends a space character. The function output_attribute_tag sends a special character used to identify attributed operators (octal ASCII code 32). The functions output_integer (t), output_operator(op) and output_attribute() are described in the following subsection.

Here are examples of encoding of some previous terms:

`A␣1␣B␣2␣1␣C␣2␣123␣D␣0␣`
`+␣3␣X␣0␣Y␣0␣3␣`
`A␣1␣?B␣2␣2␣a␣2␣b␣E␣1␣?B␣1␣0␣a␣A␣0␣1␣C␣2␣123␣D␣0␣`
where `?` is the "unprintable" character of ASCII code 26.

2.4.1 Encoding integers

ASAP defines a way to encode integers : as a sequence of characters corresponding to a writing in base 10 or in base 16.

If the integer n is negative, its encoding ends with the character `-`, then if the base is 16 `0x` is added, then the absolute value of the integer n is encoded as the sequence of its characters in base 10 in the reverse order (least significant digit first) or 16 (using `a`, `b`, `c`, `d`, `e`, and `f` as the digits) with no unnecessary terminating zeros permitted.

The number 10 becomes `01` in base 10 (`010` is not permitted) and `0xa` in base 16 (`0xa0` is not a legal encoding because of the terminating zero).
Note that 0 has a unique representation `0`.
The two bases are supported by ASAP but base 16 will certainly be the most convenient (16 being a power of 2...).

2.4.2 Encoding an array of bytes

ASAP supports the transmission of a raw sequence of bytes. Such a sequence S of length n is transmitted as the character "escape" (ASCII character number 27) followed by n encoded as an integer, followed by a blank (space) character (ASCII number 32), followed by the sequence S itself.

2.4.3 Encoding operators and attributes names

The encoding is very simple as operators and attributes are just sequences of characters containing no spaces. They are sent as such.

The valid names for operators and attributes are made up from the regular expression :

`[A-Za-z@_+-=%$#/''()[]][0-9A-Za-z@_.+-=%$#/''()[]]*`

There are some examples of valid identifiers :

```
ABeautifulIdentifier toto _fubar
```

and invalid ones :

```
0toto 1toto
```

Basically they should not contain unprintable or blank characters (space, tabulation, newline...), and should not begin with a digit.

2.4.4 What about other objects

Every mathematical object must be encoded as a term to be sent by ASAP. It is possible that for efficiency reasons special encodings must be used. For example, floating-point numbers could be sent as terms with a special operator and one argument representing the value encoded as a string (a printable representation) or as a binary value. In this later case, we must agree on one (or several) binary format (in the set of the four IEEE floating-point formats for example). It is likely that in such a situation one also needs a way to encode an array of such numbers...As the needs are not yet clear, this document does not attempt to provide a general solution. The user is free to use whatever encoding suits him inside the arrays of bytes of ASAP (he can use XDR routines to perform the encoding of C data structures, or more particular or efficient encoding).

2.5 Decoding an ASAP message

Decoding an ASAP message is quite straightforward. Such a message can be viewed as a sequence of tokens separated by space characters, each token being an operator or attribute name, an integer or an array of bytes. In particular the first character identifies the kind of token. Note that the white spaces are very important. They show where the message ends.

2.6 Predefined attributes and their semantics

All attribute names that have a meaning defined by ASAP begin with the character `@`.

- `@terminate` requests the termination of the server. Does not need an answer. The connection will no longer be valid. Could be sent on both channels.
- `@name` gives a name to the corresponding subterm (the term whose `@name` is an attribute).
- `@reference` references the subterm with the given name. If no such name is known, should signal an error in response.
- `@stream` indicates that the number of subterms is not known in advance. The incoming terms are gathered till a term is found with the attribute `@eos`. It will be the last son.
- `@eos` end of the stream of terms.
- `@error` an error occurred.
- `@RUthere`. To be sent on the urgent channel to verify that a process is still alive
- `@stillalive`. The response on the urgent channel to the `@RUthere` query...
- `@predefined` the whole term has a meaning for ASAP. Don't know yet how to use that.

Some attributes can only appear at the root of a term : `@terminate`, `@error`, `@RUthere`, `@stillalive`.

If the value of the attribute `@error` is an integer, it is to be interpreted as :

- 0 : fatal error. The process sending the message will terminate and the connection will no longer be valid
- 1 : recoverable error
- 2 : running out of memory, exiting
- 3 : running out of memory, recoverable

An example of a term containing an error :

```
_[(@error, 0)] 0
```

It uses a "dummy" operator `_` with no subterms.
The interpretation of some predefined attributes depends on being a client or a server (`@terminate` is an example)...

It is mandatory to interpret the predefined attributes before getting the "sons" of the operator (in the case of `@stream` it is of course absolutely necessary).

2.7 Restrictions

The following restrictions are enforced to help the implementations :

- attribute and operator names are limited to 256 characters in length (so that a simple preallocated buffer can be used to build them).
- number of sons and attributes are lesser than 2^{30} (so that machine integers can always be used).

Part 3

The ASAP Library

3.1 Informal discussion

3.1.1 Expressions, terms and trees

All data sent via ASAP are basically sequences of bytes that are encodings of terms (expression) that are attributed trees. There are two types of terms:

- *basic terms*, representing integers or arrays of bytes that are the leaves of the trees.
- *composite terms*, made of an operator, one or more pairs made of an attribute name and its value (another term) and a number of subterms (arguments of the operator).

A composite term can be represented as:

$$operator[(attr_1, val_1), \ldots, (attr_n, val_n)](arg_1, \ldots, arg_n)$$

(where val_i and arg_i can be such terms)

When a composite term is encoded with ASAP, the operator of the term is written followed by the number of attributes, followed by the number of arguments (so that the receiving program knows how many parts will follow). Then the encoded pairs $(attr_i, val_i)$ are sent followed by the encoded subterms.

For example, the mathematical term:

$$b^2 - 4ac$$

could be send through the channel *chan* by the following sequence of C function calls:

```
ASAPsendOp(chan, "-", 2);
ASAPsendOp(chan, "^", 2);
ASAPsendSymbol(chan, "b"); /* or ASAPsendOp(chan, "b", 0); */
ASAPsendNInt(chan, 2);
ASAPsendOp(chan, "*", 3);
ASAPsendNInt(chan, 4);
ASAPsendSymbol(chan, "a");
ASAPsendSymbol(chan, "b");
```

This term is actually encoded as the following character string :

```
-␣2␣^␣2␣b␣0␣2␣*␣3␣4␣a␣0␣c␣0␣
```

Expressions sent by ASAP may convey many different kinds of information. For example, an expression sent by ASAP from one program to another might be a request, a result of a calculation, a warning message, or graphics data. The interpretation is left to the two communicating programs. Only a small set of attributes have predefined meanings (to specify termination or interruption, unknown number of arguments or subterm sharing). It is expected that "sub-protocols" will be defined on top of ASAP (corresponding to specific layers on top of the ASAP library) to define encodings and conventions for a particular applications. In particular, the ability to send arbitrary binary data allows the definition of efficient specialized encodings on top of this library.

3.1.2 Interprocess communication

This library has been designed to allow a process (the client) to request mathematical computations from one or more processes (the servers).

The library has been designed to be used in a UNIX environment, using the *socket* interprocess communication facility. A socket is an endpoint of communication. A socket in use is usually bounded to an *address*. The nature of this address depends on the chosen *communication domain* of the socket. A characteristic property of a domain is that processes communicating in the same domain use the same *address format*.

The two domains currently implemented are the UNIX domain (AF_UNIX) (allowing communications between two processes on the same machine) and the INTERNET domain (AF_INET) (allowing connections over a network). An address in the UNIX domain is an ordinary file (a pathname). Processes communicating in the internet domain use DARPA Internet communications protocols (such as TCP/IP) and INTERNET addresses, which consist of a 32-bit host number and 32-bit port number.

The library uses *stream* sockets, as they provide sequenced, reliable, two-way connection based byte streams. For each communication there are two file descriptors corresponding to two channels: the "normal" channel is used for normal communication and the "urgent" channel is used to signal "exceptional" events (like requests of interruption) and related information.

3.2 The types used by the ASAP library

Here are the various types that are externally manipulated by the library. Their complete definition can be found in the header file `asap.h`.

`typedef enum {ASAP_SUCCESS, ...}` **ASAPstatus**;

This type is used to describe the return status of some library functions. A status different from `ASAP_SUCCESS` (or `ASAP_FINISH`) indicates a failure (see *ASAPerrorText*). See the header file `asap.h` for a complete description.

`typedef struct {...}` **ASAPserv_t**;
The type used to represent a service. It contains various information about a particular service as retrieved by *ASAPgetservbyname*.

`typedef struct {...}` ***channel_t**;
The type of the two channels of a given connection. It should be an "abstract" type as it contains the file descriptors as well as data regarding the buffering scheme and the type of the current token to be read.

`typedef struct {`**channel_t** *normal*, **channel_t** *urgent*, ...} **ASAPconn_t**;
This describes an ASAP connection and its two channels.

`typedef enum {ASAP_BIN, ASAP_INT, ...ASAP_READING_BIN, ...}` **ASAPtoken_t**;
This type identifies the kind of the current token to be read on a stream. `ASAP_INT` `ASAP_BIN`, `ASAP_OP` and `ASAP_ATTROP` identify the basic objects (binary data, integer, unattributed operator and attributed operator). `ASAP_READING_BIN` and `ASAP_READING_INT` are used to keep track of the reading of the values of unbounded size (that do not fit in a user supplied buffer). `ASAP_UNKNOWN` is used internally to indicate that the type of the current token is not yet known.

`typedef enum {ATTR_TERMINATE, ATTR_BEGIN, ATTR_END, ATTR_INTR, ...}` **ASAPpred_t**;
This type is used to represent the predefined attributes used by ASAP. These attributes can be manipulated through the functions *ASAPpredefAttr* (to recognize a predefined attribute) and *ASAPpredefName* (to get the name of a predefined attribute suitable to send it through *ASAPsendAttrName*. `ATTR_TERMINATE` requests the termination of the server process (it is used by *ASAPterminate*). `ATTR_BEGIN` and `ATTR_END` are used to implement operators with an *a priori* unknown number of sons. The operator is sent with the attribute `ATTR_BEGIN` followed by its subterms, the end is marked with a "dummy" operator (that will be ignored) attributed with `ATTR_END`. The values of these attributes are ignored (read and discarded). In the library functions a zero is used.

3.3 Services management (client side)

The following functions are normally used by a client process to obtain the connections with the needed servers. Each server can be viewed as an instance of a particular service.

3.3.1 Consulting the service database

ASAPstatus_t *ASAPgetservbyname*(**char** **name*, **ASAPserv_t** **serv*)
ASAPgetservbyname is a general function for retrieving information about a particular service; *name* is the name of the required service, *serv* is a pointer to an **ASAPserv_t** object that will be instantiated. The service is searched for in the list of files given by the (UNIX)

environment variable `ASAP_DATABASE` (whose value is "path"-like, i.e., consists of file names separated by ':' characters) or in the file `$HOME/.asapserv` if the variable is not set. This function can return `ASAP_ENOSUCHSERV` when unable to find the server in all the consulted files, `ASAP_ESYNTAX` when a syntax error was found in a database file or `ASAP_ESYS` when a system error occured.

3.3.2 Opening a connection

ASAPstatus_t *ASAPconnect*(**ASAPserv_t** *serv*, **ASAPconn_t** ∗*conn*)

ASAPconnect is a general function for opening an ASAP connection to a server corresponding to the specified service; *serv* describes the service (obtained by *ASAPgetservbyname*, and *conn* will be instantiated.

3.3.3 Closing a connection

ASAPstatus_t *ASAPterminate*(**ASAPserv_t** *serv*, **ASAPconn_t** *conn*)

This terminates the connection *conn* to the service *serv*. It is required to free critical system resources (like file descriptors and to remove the files implementing the UNIX domain socket). The server process will normally be terminated (it depends on the type of service, some can choose to continue listening to new connection requests). This is not an "emergency" exit as it will first try to send any "remaining" data (but not to read any available data that will be discarded).

3.4 Connections management (server side)

To be launched by the mechanism of the *ASAPconnect* function, server programs should follow some simple conventions to be able to retrieve the address used to communicate on their "command line". In the case of an INTERNET domain connection, one argument is `--machine`, followed by the INTERNET name (or address in the "dot" format) of the machine on which the client runs (*ASAPconnect* will always use the "dot" format to speed up things) A `--port` argument is mandatory to give the port on which the client will listen (this is the port number p of the normal channel; the urgent channel will be port $p+1$). In the case of a UNIX domain connection, the `--socket` argument is followed by the base name f of the files implementing the two sockets: the normal channel will be at the address f`n` and the urgent one at f`u`.

A server should be ready to accept both kind of connections. Two functions are then provided by the library in both communication domains to establish the connection to the client.

3.4.1 Opening a connection

ASAPstatus_t *ASAPlocalConnect*(**char** ∗*socketname*, **ASAPconn_t** *conn*)

Opens a local connection (in the UNIX domain) to the client.

ASAPstatus_t *ASAPremoteConnect*(**char** ∗*machine*, **int** *port*, **ASAPconn_t** *conn*)
Opens a remote connection to the client (on the given machine); *machine* can be a name or an address in "dot" format.

3.4.2 Closing a connection

ASAPstatus_t *ASAPserverEnd*(**ASAPconn_t** *conn*)
Closes the connection corresponding to the client *conn*. This closes the file descriptors.

3.5 Sending and receiving

The following functions are used by both clients and servers to exchange the data. The channels used by these functions are found in the **ASAPconn_ t** structure (as their `normal` and `urgent` fields) returned by the above functions (*ASAPlocalConnect* and *ASAPremoteConnect* for a server and *ASAPconnect* for a client). The library uses a buffering scheme for efficiency. The user cannot thus predict when the data will be actually sent (and received) without explicitly calling the *ASAPflush* function on a channel. Note that in the case where the data to be sent are larger than the internal buffers, they will be sent directly (bypassing the buffering).

The functions for sending and receiving terms can return `ASAP_BROKEN` indicating that the connection has been broken (unexpected death of the client or the server).

3.5.1 Sending

ASAPstatus_t *ASAPsendNInt*(**channel_t** *chan*, **int** *i*)
Sends the integer *i* (a "natural" C integer) on the channel *chan*.

ASAPstatus_t *ASAPsendInt*(**channel_t** *chan*, **char** ∗*s*, **int** *len*)
Sends an integer represented as the character string *s* of length *len*. The integer should be written in decimal or hexadecimal (using `abcdef` as the digits and `0x` as a prefix) with the digits reversed (least significant first). So that, for example, 1034 will be `0xa04` or `4301` and −34234 will be `0xab58-` or `43243-`.

ASAPstatus_t *ASAPsendBinary*(**channel_t** *chan*, **cha r** ∗*data*, **int** *len*)
Sends the specified sequence of bytes (*data*) on the channel *chan*. No interpretation is provided for this sequence.

ASAPstatus_t *ASAPsendOp*(**channel_t** *chan*, **char** ∗*op*, **int** arity)
Sends an operator whose name is *op* (a C character string, ending in 0) and arity *arity* with no attributes.

ASAPstatus_t *ASAPsendAttrOp*(**channel_t** *chan*, **cha r** ∗*op*, **int** *arity*, **int** *nbattrs*)
Sends an operator whose name is *op* (a C character string, ending in 0) and arity *arity* with *nbattrs* attributes.

ASAPstatus_t *ASAPsendAttrName*(**channel_t** *chan*, **c har** ∗*name*)
Sends an attribute of name *name* over the channel *chan*.

ASAPstatus_t *ASAPflush*(**channel_t** *chan*)
Ensures that any buffered data is sent over the channel *chan*.

ASAPstatus_t *ASAPsendSymbol*(**channel_t** *chan*, **cha r** ∗*symbol*)
Sends a term with operator *symbol* (a C character string), arity 0 and no attributes.

ASAPstatus_t *ASAPsendPredefAttr*(**channel_t** *chan*, **ASAPpred_t** *attr*)
Sends the given predefined attribute *attr* on channel *chan*.

ASAPstatus_t *ASAPsendListOp*(**channel_t** *chan*, **cha r** ∗*op*)
Sends an operator with unknown number of sons. The end will be indicated by calling the function *ASAPsendEndOp*. Note that there is no direct means to send such an operator with other attributes (you have to make an explicit call to *ASAPsendAttrOp* and *ASAPsendAttrName*.

ASAPstatus_t *ASAPsendEndOp*(**channel_t** *chan*)
Terminates the list of the subterms of the last operator attributed with `ATTR_BEGIN`.

3.5.2 Receiving

It is a little bit more complicated than sending due to allocation problems from the potentially unbounded size of integers and binary data. The user should thus provide a buffer to read these objects (by the *ASAPgetInt* and *ASAPgetBinary* functions). If the buffer is not big enough the rest of the data is obtained with particular functions (*ASAPgetMoreInt* and *ASAPgetMoreBinary*). Note that the size of an operator or attribute name is limited to `MAXOPLEN` and `MAXATTRLEN` (these lengths do not include the 0 byte used to mark the termination of the strings returned by *ASAPgetOp* or *ASAPgetAttrName*).

ASAPtokenType_t *ASAPgetTokenType*(**channel_t** *chan*)
Returns the type of the next token to be read on *chan*. The following functions that retrieve tokens return `ASAP_EBADTYPE` if requested to read a token whose type does not match the required type (as for example when *ASAPgetInt* is called when an operator is actually the next token on the channel).

ASAPstatus_t *ASAPgetInt*(**channel_t** *chan*, **char** ∗*buf*, **int** *len*, **int** ∗*read*)

The argument *buf* is a buffer that will be filled with the characters representing the integer present on the channel *chan* (there will be no null character at the end). If the integer fits in the buffer, the function returns `ASAP_FINISH` (instead of `ASAP_SUCCESS`); *read* will be the number of bytes actually put into *buf*. The function *ASAPgetTokenType* will return `ASAP_READING_INT` while the integer is not fully read.

ASAPstatus_t *ASAPgetMoreInt*(**channel_t** *chan*, **char** ∗*buf*, **int** *len*, **int** ∗*read*)
To be used to read what remains of the current big integer. It should be iterated until it returns `ASAP_FINISH` (instead of `ASAP_SUCCESS`); *read* will contain the number of bytes read.

ASAPstatus_t *ASAPgetNInt*(**channel_t** *chan*, **int** ∗*i*)
The argument *i* will be the value of the integer present on the channel *chan*. Returns `ASAP_ERANGE` if the integer cannot fit in an **int** value.

ASAPstatus_t *ASAPgetBinary*(**channel_t** *chan*, **char** ∗*buf*, **int** *len*, **int** ∗*read*)
Reads a binary data in the user supplied buffer *buf* of length *len*. If the buffer is too small `ASAP_SUCCESS` is returned, else `ASAP_FINISH` is. The variable *read* will be set to the number of characters read. The function *ASAPgetTokenType* will return `ASAP_READING_BIN` while the binary data is not fully read.

ASAPstatus_t *ASAPgetMoreBinary*(**channel_t** *chan*, **char** ∗*buf*, **int** *len*, **int** ∗*read*)
Reads what remains of the binary data. Should be iterated until it returns `ASAP_FINISH`. The variable *read* will contain the number of bytes r ead.

ASAPstatus_t *ASAPgetOp*(**channel_t** *chan*, **char** ∗*op*, **int** ∗*arity*)
Gets the operator *op* of the term and its arity *arity*. It returns `ASAP_EBADTYPE` if the operator has attributes.

ASAPstatus_t *ASAPgetAttrOp*(**channel_t** *chan*, **char** ∗*op*, **int** ∗*arity*, **int** ∗*nbattr*)
Gets the operator *op* of the term, its arity *arity* and number of attributes *nbattr*. It returns `ASAP_EBADTYPE` if the operator has no attribute.

ASAPstatus_t *ASAPgetAttrName*(**channel** *chan*, **char** ∗*attr*)
Returns the name *attr* of the attribute.

ASAPpred_t *ASAPpredefAttr*(**char** ∗*attr*)
Tests if the character string *attr* represent a predefined attribute. If true, it returns this attribute (as a member of the **ASAPpred_t** enumeration type). `ATTR_UNKNOWN` is returned if false.

3.5.3 Managing exceptional conditions

Exceptional conditions are normally received (and sent) on the urgent channel, as legal terms. The preceding functions can thus be used to send on and read from this channel (which is accessed as the `urgent` field of an **ASAPconn_t** structure. Due to their asynchronous nature, the normal computation should be interrupted to read the urgent channel and take whatever action is required. This behaviour can be obtained by enabling the SIGIO signal on the urgent descriptor (as is now done by the ASAP library).

Some experiments are needed to understand the burden that such a scheme place on the programmer. At least we should provide a default handler and a function to install "default" behavior for the common exceptional conditions. A first very crude suggestion would be to set a global flag.

3.5.4 Miscellaneous

To cope with possible problems of memory management, the library uses as a memory allocator the value of the variable *ASAPmalloc* (initially the system "natural" memory allocator *malloc*). For the moment, this allocator is only used to obtain **ASAPconn_t** structures (containing the buffers necessary for each channel) and **ASAPserv_t** structures (inside *ASAPgetservbyname*).

`extern` **char** $*(*ASAPmalloc)($**unsigned** *size*$)$

Should return `NULL` if it fails. If it succeeds, it returns a pointer to a block of at least *size* bytes, which is appropriately aligned.

char *ASAPversion*[]

a string identifying the current version of the library.

char $*$*ASAPpredefName*(**ASAPpred_t** *attr*)

Returns the character string that is the representation of the predefined attribute *attr* (needed for calling the *ASAPsendAttrName* function).

char $*$*ASAPerrorText*(**ASAPstatus_t** *status*)

Returns a pointer to a message text describing the given *status*.

char $*$*ASAPperror*(**ASAPstatus_t** *status*, **char** $*$*msg*)

Prints (using the standard error file descriptor) the message *msg* followed by a colon ':' surrounded by spaces, followed by a description of the error associated with *status* (using *ASAPerrorText*), followed by a newline character. When *status* is `ASAP_ESYS` the text obtained by *ASAPerrorText* is followed by a message describing the system error (as can be obtained with the *perror* standard function).

3.6 The service database

The service database allows to establish a correspondence between a service name and a way to obtain a server providing this service. We define two kinds of services corresponding to servers that need to launched by the client and servers that are already running waiting for connections.

3.6.1 The format of service database files

Here is a very simple example:

```
macaulay          /usr/local/bin/macaulayserv
macaulay-psyche   psyche.inria.fr:/usr/local/bin/macaulayserv
gbmodulo          ganesa:/usr/local/bin/macaulayserv
```

Everything after a '`#`' is taken as a comment (and ignored) until the end of the line. White lines are ignored as well. Each line as one of the following format:

⟨service name⟩ ⟨machine name⟩`:`⟨command line⟩
⟨service name⟩ ⟨command line⟩
⟨service name⟩ ⟨machine name⟩`:`⟨integer⟩
⟨service name⟩ `:`⟨integer⟩
⟨service name⟩ `!`⟨pathname⟩

where ⟨service name⟩ is the name of the service (valid values for this name are given by the regular expression `[0-9A-Za-z@_.+-=%$/''`]+), ⟨machine name⟩ is a machine name (whatever it means, it should not contain white space, nor '`#`' nor '`!...`'), ⟨command line⟩ is a command line that will be interpreted by a command interpreter (locally or remotely) and ⟨pathname⟩ is a valid pathname.

The first form indicates that the service is started by issuing the given command line on the given machine. The second one indicates that the service is started on the local machine by executing the given command line. The third form indicates that the required service is listening the given (TCP) port number on the given machine. The fourth that it listens on the local machine. The last indicates that it listens on the UNIX domain.

Services of the first kind will be invoked, adding the arguments `--machine` and `--port` to their command lines, with as respective values the name of the local machine (fully qualified INTERNET name or IP address) and a port number to connect to (the normal channel; the urgent one will be bound to $p+1$). Services of the second kind will be invoked, adding the argument `--socket` whose value is a pathname *path*. The service must then connect to the UNIX domain address *path*`u` for the urgent channel and *path*`n` for the normal one. Services of the other three kinds are assumed to be servers that are running already, and will not be invoked explicitly. Moreover they should conform to the same convention about port numbers and path names, i.e., the specified port numbers p correspond to the "normal" channel, while the associated urgent channels are bound to $p+1$. Similarly The actual pathnames associated to UNIX domain connections are obtained by adding the suffixes '`n`' and '`u`' to the given pathnames.

3.7 Using ASAP library functions in your C programs

Just include the header file `asap.h` in any source file that need to use functions from the ASAP library (and use the `-I` argument to locate this file). Compiling the library should have produced an archive named `libASAP.a` that you can link with `-lASAP` with the correct library path (set with the `-L` option) (of course, your actual options depend on the C compiler and link editor you use...).

The ASAP library has been compiled on SUN and DEC machines (SPARC based and MIPS based) running SUNOS 4.? and ULTRIX. The library should be fairly portable to every UNIX derived system with sockets. It is written in "old C" but the header file `asap.h` includes the prototypes of the library functions for any "standard C" compiler (whose preprocessor defines the symbol `__STDC__`).

3.8 A simple example

This simple example of the use of the ASAP library is extracted from a client and a server used for testing purpose. The client will launch a server corresponding to a service given on its command line. It will then send various kind of terms, checking the answers from the server. Some error checking has been omitted in this version.

3.8.1 The client

```
#include <stdio.h>
#include "asap.h"


main(argc,argv)
    int argc;
    char *argv[];
{
    ASAPserv_t serv;
    ASAPconn_t conn;
    ASAPstatus_t err;
    extern void test_integer();

    if (argc != 2) {
       printf("usage: client service\n");
       exit(0);
    }
    if ((err = ASAPgetservbyname(argv[1], &serv)) != ASAP_SUCCESS) {
        ASAPperror(err, "cannot find server");
        exit(1);
```

```
    }
    if ((err = ASAPconnect(serv,&conn)) != ASAP_SUCCESS) {
       ASAPperror(err, "cannot connect");
       exit(1);
    }
...

    /* test integers and terminate */
    test_integer(conn, 100);
    ASAPterminate(conn);
    exit(0);
}
void test_integer (conn, max)
   ASAPconn_t conn;
   int max;
{
   int n, r, nb, i;
   char buf[BUFSIZ];
   ASAPstatus_t err;

   for (i = 0; i < max; i++) {
      n = really_random(INT_RANGE);
      ASAPsendNInt(conn.normal, n);
      if ((err = ASAPflush(conn.normal)) != ASAP_SUCCESS) {
         ASAPperror(err, "error in test_integer");
         terminate(conn);
      }
      if (ASAPgetTokenType(conn.normal) != ASAP_INT) {
         printf("error in test_integer, expecting an integer\n");
         terminate(conn);
      }
      if (ASAPgetInt(conn.normal, buf, BUFSIZ, &nb) != ASAP_FINISH) {
         printf("error in test_integer: ASAPgetInt\n");
         terminate(conn);
      }
      r = decodeInt(buf, nb);
      if (r != n + 1) {
         printf("test_integer: test failed\n");
         terminate(conn);
      }
   }
```

```
}
```

3.8.2 The server

```
#include <sys/param.h>
#include <stdio.h>
#include <ctype.h>
#include "asap.h"

main (argc, argv)
int argc;
char *argv[];
{
    char machine[MAXHOSTNAMELEN];
    char socketname[MAXPATHLEN];
    int port;
    int machineflag = 0;
    int socketflag = 0;
    int portflag = 0;

    ASAPconn_t conn;

    int i;

    for(i = 1; i < argc; i++) {
        if (strcmp(argv[i],"--port") == 0) {
            portflag = 1;
            if (i == argc - 1) fatal("expecting a port number");
            port = atoi(argv[i+1]);
            i++;
        } else if (strcmp(argv[i],"--machine") == 0) {
            machineflag = 1;
            if (i == argc - 1) fatal("expecting a machine name");
            strcpy(machine, argv[i+1]);
            i++;
        } else if (strcmp(argv[i], "--socket") == 0) {
            socketflag = 1;
            if (i == argc - 1) fatal("expecting a socket name");
            strcpy(socketname, argv[i+1]);
            i++;
        } else
            fatal("unknown option");
```

```
     }

    if (socketflag) {
        if (machineflag || portflag) fatal("incompatible options");
    }
    else if (! (machineflag && portflag)) fatal("lacking one option");

    if (socketflag) ASAPlocalConnect(socketname, &conn);
    else ASAPremoteConnect(machine, port, &conn);

    test(conn);
}
void test (conn)
   ASAPconn_t conn;
{
    for(;;)
    {
       switch (ASAPgetTokenType(conn.normal))
       {
       case ASAP_INT:
          test_int(conn);
          break;
       case ASAP_BIN:
          test_binary(conn);
          break;
       case ASAP_OP:
          test_op(conn);
          break;
       case ASAP_ATTROP:
          test_attrop(conn);
          break;
       default:
          fprintf(stderr, "token type...\n");
          exit(1);
       }
    }
}

void test_attrop(conn)
    ASAPconn_t conn;
{
```

```
   char op[MAXOPLEN + 1];
   char attr[MAXATTRLEN + 1];
   int arity;
   int nbattrs;
   ASAPpred_t predef;

   ASAPgetAttrOp(conn.normal, op, &arity, &nbattrs);
   if (nbattrs == 1) {
      ASAPgetAttrName(conn.normal, attr);
      if ((predef = ASAPpredefAttr(attr)) == ATTR_TERMINATE) {
         exit(1);
      }
   }
   fprintf(stderr, "unexpected attribute\n");
}
```

Bibliography

[Kajler1] N. Kajler : *Building a Computer Algebra Environment by Composition of Collaborative Tools*, *DISCO'92*, Bath, GB, April 1992.

[Kajler2] N. Kajler : *Environnement Graphique Distribué pour le Calcul Formel*, Thèse, Mars 1993, Université de Nice-Sophia Antipolis.

[Doleh] Y. Doleh, P.S. Wang. *SUI : A system Independent User Interface for an integrated Scientific Computing Environement*, *ISSAC'90*, pages 88-94, Tokyo, August 1990.

[Stillman] M. Stillman, D. Bayer : *Macaulay User Manual*, August, 1989.

[Mourrain] B. Mourrain : *MICMAC : Maple Incorporated Communications with Macaulay*, December, 1992.

[Wolfram] : Wolfram Research : *MathLink Reference Guide* 1991-1992.

[Riflet] J.M. Rifflet : *La communication sous UNIX* 2nd edition

[Vorkoetter] S.Vorkoetter : *Open Math proposal*, preliminary document, december 1993.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LES NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399